An Architecture Framework for Application-Managed Scaling of Cloud-Hosted Relational Databases

Liang Zhao, Sherif Sakr, Liming Zhu, Xiwei Xu, Anna Liu

1National ICT Australia (NICTA)
†School of Computer Science and Engineering
University of New South Wales, Australia
{firstname.lastname}@nicta.com.au

ABSTRACT
Scaling relational database in the cloud is one of the critical factors in the migration of applications to the cloud. It is important that applications can directly monitor fine-grained scaling performance (such as consistency-related replication delays and query-specific response time) and specify application-specific policies for autonomic management of the scaling. However, there is no general mechanism and reusable framework and infrastructures to help this. The current facilities in cloud-hosted relational databases are also very limited in providing fine-grained and consumer-centric monitoring data. The situation is exacerbated by the complexity of the different underlying cloud technologies and the need to separate scaling policy from business logic. This paper presents an architecture framework to facilitate a consumer-centric, application-managed autonomic scaling of relational databases in cloud. The architecture framework includes a new consumer-centric monitoring infrastructure and customisable components for sensing, monitoring, analysing and actuation according to application-level scaling policies without modifying an existing application. We evaluated our framework using a modified Web 2.0 application benchmark. The results demonstrate the framework’s ability to provide application-level flexibility in achieving improved throughput, data freshness (different levels of consistency) and monetary saving.

Categories and Subject Descriptors
D.2.11 [Software Engineering]: Software Architectures—middleware; H.2 [Database Management]: Miscellaneous—replication delay, performance measurement

General Terms
Design, Performance

Keywords
actuator, Amazon EC2, application-managed, cloud, framework, monitor, sensor

1. INTRODUCTION
The cloud computing technology has opened up new avenues for deploying novel applications which were not economically feasible in a traditional enterprise infrastructure setting. For example, the cloud has become an increasingly popular platform for hosting software applications in a variety of domains such as e-retail, finance, news and social networking. Thus, we are witnessing a proliferation in the number of cloud-hosted applications with a tremendous increase in the scale of the data generated as well as being consumed by such applications. Cloud database systems power these applications form a critical component in the software stack of these applications.

In practice, there are different options for deploying database systems on cloud environments. One approach is to explicitly architect the application for cloud and leverage self-scaling cloud-provided storage systems provided by the cloud vendors [1]. This approach requires significant re-engineering of an existing application or greenfield development, which is often not feasible. The application also has very limited visibility into the scaling infrastructure and control over when and how to scale according to application specific requirements and policies. Another approach is the cloud-provided database systems in which a third party service provider hosts a relational database as a service. Such services allow the pay-as-you-go model for using traditional relational databases but with feature restrictions, monitoring and scaling control limitations introduced by the service provider [2].

The third approach is to take an existing application using a relational database designed for a conventional data center, and then port both the application and the database to the cloud [2, 3, 4]. The migrated database therefore turns to be cloud-hosted database systems, which can take advantages of geographic distribution at a lower cost, in comparison with building conventional data centers across continents that can be only afforded by big companies. Such migration usually requires minimum changes in the architecture or the code of the deployed application [5]. When this is done, the application tier can easily take advantage of the elasticity and scale provided by the cloud, as well as retain its original ways of using the database and have full control in dynamically allocating and configuring the physical resources of the database tier [2, 3, 4]. However, the data management layer, being stateful, faces more issues, such as complexities and challenges in fine-grained monitoring and management of the scaling, especially for the consumer side. In the context of cloud, a fine-grained monitoring and management must take performance and momentary cost into consideration. An ad-

justable check period is necessary for a fine-grained method to suit difference host performances.

The first challenge comes from the complexity and limited consumer-centric monitoring facilities from a large number of different cloud providers. Many research provide ways of monitoring and optimising scaling strategies at runtime [7, 9] but largely targeting cloud providers. Our approach introduces a new consumer-centric monitoring infrastructure to collect more fine-grained scaling-related performance data. The second challenge comes from the nature of applications being migrated to cloud. While traditional transactional data management applications (e.g. banking, stock trading) usually require microsecond precision for their read operations, eventual consistency model [7] is more amenable to many Web 2.0 applications (e.g. social network applications) which could be more tolerant with a wider window of data staleness (replication delay) [9]. Several very large Web-based systems such as Amazon, Google and eBay have relied on the eventual consistency model for managing their replicated data over distributed data centers [7, 9]. However, there are no general mechanisms for applications to easily monitor, specify and manage, according to application-specific characteristics and goals, to what extent inconsistencies across cloud-hosted databases and replicas can be tolerated. Our approach introduces a general mechanism with a simple XML-based language for policies. The third challenge comes from the architect-level reuse and modularity needs of effectively developing such fine-grained consumer-centric monitoring, policy specification, analysis and actuation [3]. Our approach introduces an architecture framework with customisable and extensible components for sensing, monitoring, analysing and actuation in the cloud-hosted relational database environment.

Scaling database is inherently coupled to the access, usage and semantics of the data that are being replicated, sharded and clustered. This environment is characterised by high latency communication between data centres and significant fluctuations in the performance of underlying virtual machines [7, 9]. This demands an adaptive application-managed and consumer-centric solution for scaling databases in (multi) data centre cloud platforms. The solution presented in this paper is an architecture framework to facilitate such application driven autonomic management of database scaling, such as replication, in cloud. Our work applies the concepts in autonomic computing [7] by including components for sensing, monitoring, analysing and actuation according to specified application-level policies without modifying an existing application. By applying an architecture-based adaptation approach [5], the architecture framework provides reusable infrastructures for monitoring and allows further customisation of what aspects of the system to monitor, what conditions should trigger what types of scaling action.

The major contributions of the architecture framework include:

1. a new consumer-centric monitoring infrastructure for cloud-hosted relational databases,
2. an architecture framework with customisable components for building application specific scaling behaviours for cloud-hosted relational databases,
3. the support of different cloud vendors using the same set of component interfaces and policy languages,
4. the separation of database scaling logic from business logic so existing applications do not have to be modified, and
5. a low overhead in terms of performance.

We elaborate our approach through traditional replication-based scaling-out but the the framework is also applicable to other scaling techniques such as sharding, scaling-up and hybrid approaches. The current replication-related part of the framework implementation allows the keeping of several replicas of the database in different data centres to support the different availability, scalability and performance goals. The framework also allows the specification of different SLA data freshness for the underlying database replicas. In particular, the framework allows the specification of an SLA of data freshness for each database replica and continuously monitor the replication delay of each replica so that once a replica violates its defined SLA, the framework automatically activates another database replica at the closest geographic location in order to balance the workload and re-satisfy the defined SLA.

The remainder of this paper is structured as follows. Section 2 gives a brief overview about database replication scaling-out strategies in cloud environments. Section 3 introduces the architecture framework, its new monitoring infrastructure and customisable components. Section 4 represents the evaluation results. Section 5 summarises the related work before we conclude the paper in Section 6.

2. DATABASE REPLICATION IN THE CLOUD

Database replication [3] is a well-known scaling-out strategy to achieve performance goals in the data management world. In practice, two database replication architectures are commonly used: the multi-master replication and the master-slave replication. The multi-master replication allows each replica to serve both read and write requests. In this architecture, write-write conflicts are resolved by the replication middleware so that each replica executes write transactions in the same sequence. On the other hand, the master-slave replication is better at improving read throughput because read requests are served by slaves while all the write requests are only served by the master. The replication middleware in this case is then in charge of passing writesets from the master to slaves to keep the database replicas up-to-date. The write-write conflicts are completely resolved on the master side. We elaborate our general approach in the master-slave replication context as it is the most commonly used approach by many Web 2.0 applications.

The replication configuration and architecture exposes how read and write requests are assigned across replicas while the synchronisation choice reveals how data is committed across replicas. For example, the synchronous replication will block a response to the client until the write transaction is committed on the updated replica and writesets are propagated to all other replicas. By doing so, it guarantees all replicas are consistent during the time. However traversing all replicas may incur high latency on write transactions. In addition, the availability of the system may get affected if unreachable replicas due to network partitioning cause a suspension of the synchronisation. The asynchronous replication sends a response once the write transaction is committed on the updated replica. The writesets will then propagated to all other replicas at a later time. It avoids potentially high write latency over networks in exchange of stale data. Furthermore, if the updated replica goes offline before duplicating data, data loss may occur. Due to the existence of replication delays, read requests on database replicas are not expected to get consistent results all the time but it will guarantee an eventual consistency [7].

Florescu and Kossmann [7] have argued that in large-scale web
applications, the requirement to provide very high read and write availability for all users is often more important than taking the ACID paradigm as the gold standard for data consistency. The strong consistency requirement that was hard and expensive in traditional databases is becoming relaxable and optimisable goals in cloud-based database systems thus the need for the specification of these goals and some degree of autonomic management of their achievement.

3. THE ARCHITECTURE FRAMEWORK

Figure 1 shows an overview of the monitoring infrastructure and the key components within the architecture framework and its relationship with the applications and the database being scaled (replicated in this case). In this architecture, monitoring infrastructure contains the reusable non-intrusive interceptors, basic probes and data collection facilities. Thus, our solution does not require any code modification on the consumer applications. For database replication monitoring infrastructure, we used a database proxy as the interceptor for non-intrusively monitoring database requests/ responses. We use a heartbeat mechanism for probing the replication delay and a heartbeat database for collecting relevant data. Different sensor components can be created and customised for collecting raw data which can then be filtered and calculated by the highly configurable monitor components for specific performance data of interests. The analyser component is responsible for analysing situations by continuously checking the monitored performance data against its associated application-defined SLA. If any violation is detected, the actuator component can trigger a scaling action based on a strategy. For database replication delay, the sensors collect probing transaction ids and local timestamps at various locations and then the monitors filter through the relevant data for a specific period and calculates the various replication delays. The analyser then continuously check the data freshness SLAs, and if necessary, asks the actuator to create or destroy a replica at a location. Application-specific SLAs and replication policies are specified in an XML-based policy language and stored in a repository.

The design of all the components follows two principles, function-extendible and application-independent. Function-extendible means any new sensing and monitoring requirements (e.g. throughput or response-time or new calculation formulas), new analyser algorithms or new types of actuation controls can be easily added through configuration or extension of the basic components. Common actions (e.g. starting a new slave replica) can be selected from a list of available actions in the actuation components. Application-independent means that all customisation, extension and new objectives are added with no code-level modification to existing applications. However, some new tools, database configurations and plug-ins may need to be employed or enabled for the existing application and database at the system level to work with our approach.

In this paper, we focus on the implementation of replication and consistency management part of the framework. We also demonstrate the independence of the framework over cloud vendors by integrating it with a database-focused Cloudstone implementation. The detailed use of tools, database configuration and plug-ins are addressed in Section 4.2.

In the following subsections we describe the implementation details for each of the main components of our architecture framework. The framework provides some default implementation of these reusable components so developers do not have to develop them from scratch. These components can be further extended if required or directly used through the flexible policy language.

3.1 Monitoring Infrastructure

The new monitoring infrastructure is for collecting important consumer-centric scaling-related performance data. In the case of database replication, one such piece of data is the replication delay between the master database and each database replica. Our monitoring infrastructure allows the collection of such data without the instrumentation of existing applications.

In order to achieve this, we rely on a database proxying mechanism which provides the ability to intercept database requests to the underlying databases and return the results from those requests transparently to the client program without the need of having any database drivers installed. A database proxy software is a simple program that sits between the client application and the database server that can monitor, analyse or transform their communications if necessary. Such flexibility allows for a wide variety of uses such as load balancing, query analysis and query filtering. One problem with the current proxy service is a possibility of a single point of failure - if the proxy goes down, the whole database system might not be accessible. A possible solution is to include multiple proxies in parallel. Instead of using one proxy, users are assigned with a list of proxies and randomly uses one for accessing at anytime.

We also created a Heartbeats database in the master and each synchronised slave database server. Each Heartbeats database maintains a “heartbeat” table with two fields: an id and a timestamp. A database request to insert a new record with a global id and a local timestamp is periodically sent to the master. Once the inserted record request is replicated to the slaves, every slave re-executes the request by committing the same global id and its own local timestamp. The update frequency of records in the master is configurable (through the sensor components), named as heartbeat interval in millisecond unit (the default configuration of the heartbeat interval is set to be 1000 milliseconds). The global id and the local timestamps are used for calculating the replication delays (also known as data staleness window) for each replica, which is done by

2http://code.google.com/p/clouddb-replication/
In the context of database replication, there are usually two types of adaptations, scaling out and scaling in. In general, adding a new database replica involves extracting database content from an existing replica and copying that content to a new replica. In practice, the time of executing these operations mainly depends on the database size. To provision database replicas in a timely fashion, it is necessary to periodically snapshot the database state in order to minimise the database extraction and copying time to that of only the snapshot synchronisation time. Apparently, there is a tradeoff between the time to snapshot the database, the size of the transactional log and the amount of update transactions in the workload. In our framework, this trade-off can be controlled by application-defined parameters in the actuator module. This tradeoff can be further optimised by applying recently proposed live database migration techniques [?, ?].

The actuator component also keeps a list of common and reusable actions. New actions can be added as needed. For database replication, our framework interacts with databases and the database proxy. Example actions include starting a new slave replica and restarting a proxy.

3.6 Policy Specification and Repository

XML schemas were defined to describe the SLA policies associated with the various components.

For database replication, XML-based policies define the replication delay SLA of each replica, as well as other replica-related information (e.g. address, location and current status). Thus the analyser component can start a nearby new replica to scale when a replica exceeds replication delay for a certain period of time. We show an example snippet below.

```
<replications>
  <replication sla="1270" address="..." zone="us-west-1b" status="Paused" />
</replications>
```

Another example snippet below shows the SLA policies for query-specific operations so that the response time of an operation can be monitored. The response time of the operation is a sum of related queries, which are pattern matched from proxy logs on the fly. If the response time of an operation exceeds the SLA for a certain amount of times, then the analyser component triggers a scaling action.

```
<operations>
  <operation name="TagSearch" sla="120">
    <query order="0" pattern=" select ... from Tags ... ">
    <query order="1" pattern=" select ... from Users ... ">
  </operation>
</operations>
```

4. EXPERIMENTAL EVALUATION
4.1 Workload design
The Cloudstone benchmark represents a typical Web 2.0 social events calendar that allows users to perform individual actions (e.g., browsing, searching and creating events) as well as social actions (e.g., joining and tagging events)\[^2\]. The original benchmark has been designed as a performance measurement tool for all components of a Web 2.0 application including both the web application layer and the database layer. We modified the benchmark to create a database-focused version which emphasised the database tier \[^3\]. In particular, we re-implemented the business logic of the application in a way that a user’s operation can be processed directly at the database tier without any intermediate interpretation at the web server tier. We also included a connection pool (i.e. DBCP\[^1\]) to reuse connections that have been released by other users who have completed their operations in order to save the overhead of creating a new connection for each operation.

4.2 Experiment design and setup
Figure 2 illustrates the setup of our experiments in Amazon EC2 platform. The experiment setup is a multiple-layer implementation. The first layer represents the Cloudstone benchmark which generates varying workload of database requests with fixed read/write ratio. The benchmark is deployed in a large instance to avoid any overload on the application tier. The second layer hosts the MySQL Proxy and our application-managed autonomic replication built using our framework. The third layer represents the database tier (MySQL Cluster) that consists of all the database replicas where the master database receives the write operations from the load balancer and then it becomes responsible for propagating the writesets to all the slave replicas. The master database runs in a small instance so that an increasing replication delay is expected to be observed along with an increasing workload \[^2\]. The master database is closely located to the benchmark, the load balancer and our framework. They are all deployed in the location of us-west. The slave replicas are responsible for serving the read operations and updating the writesets. They are deployed in three regions, namely: us-west, us-east-1e and eu-west. All slaves run in small instances for the same reason of provisioning the master instance.

We implemented a set of experiments in order to evaluate the effectiveness of our approach in terms of its effect on the replication delay for the underlying database replicas. In the experiments, we fix the monitor interval (\texttt{intvl_mon}) to 120 seconds and adjusts the SLA of replication delay (\texttt{delay_sl}) to 1000 milliseconds. The experiment runs for a period of 3000 seconds with a starting workload of 220 concurrent users and database requests with read/write ratio of 80/20. The workload gradually increases in steps of 20 concurrent users every 600 seconds so that the experiment ends with a workload of 300 concurrent users. The experiment deploys 6 replicas in 3 regions where each region hosts two replicas: the first replica is an active replica which is used from the start of the experiment for serving the database requests of the application while the second replica is a hot backup which is not used for serving the application requests at the beginning of the experiment but can be added by the actuation component, as necessary, when triggered by the analyser component.

4.3 Monitoring infrastructure configuration
On the database tier, we are using a MySQL Cluster which is deployed in Linux environment where every database replica contains an Olio database of the Cloudstone benchmark and a Heartbeats database for monitoring the replication delay. We also used a plugin software for achieving high-precision measurement of the replication delay. It consists of a user defined time/date function at microsecond resolution and a forced clock synchronisation with multiple time servers every second via NTP\[^6\] protocol. The heartbeat and monitoring intervals will be configured in the corresponding sensing and monitoring components.

MySQL Proxy\[^4\] with read and write split enabled resides in the middle between the benchmark and the database replicas and acts as a load balancer to forward write and read operations to the master and slaves correspondingly. Although the MySQL Proxy is released by the MySQL community in alpha version, it appears to be performance efficient and low overhead during our experiments. The time overhead of the current proxy (MySQL Proxy) is about 400 microsecond.

The monitoring infrastructure will also evaluate the round-trip component (\texttt{delay_rtt}) of the replication delays SLA (\texttt{delay_sl}) for the database replicas in the three geographical regions of our deployment by running \texttt{ping} command every second for a 10 minutes period. The resulting average three round-trip times (\texttt{delay_rtt}) are 30, 130 and 200 milliseconds for the master to slaves in \texttt{us-west}, \texttt{us-east}, and \texttt{eu-west} respectively.

4.4 Architecture components configuration and instantiation
Most of the component configuration and instantiation is straightforward. In our monitoring component instantiation, the replication delay calculation is triggered by the corresponding sensor thread after fetching the records. It can be triggered in other ways. In

\[^1\]http://radlab.cs.berkeley.edu/wiki/Projects/Cloudstone
\[^2\]http://code.google.com/p/clouddb-replication/
\[^3\]http://commons.apache.org/dbcp/
\[^4\]http://www.ntp.org/
\[^5\]https://launchpad.net/mysql-proxy
the general case of assuming there are $n$ and $k$ local timestamps in total in the master array ($timestamps_m$) and the slave array ($timestamps_s$), the slave’s $i^{th}$ replication delay $delay[i]$ is computed as follows:

$$delay[i] = timestamps_s[i] - timestamps_m[i]$$

where $i \leq k = n$ and the master and the slave databases are fully synchronized. In the case of $k < n$ there is a partial synchronization between the master and the slave databases which composes of consistent part and inconsistent part, the computation of the $delay[i]$ of the slave breaks into two parts: The delay of the consistent part with $i \leq k$ is computed using Equation (1). The delay of the inconsistent part with $k < i \leq n$ is computed as follows:

$$delay[i] = timestamps_s[k] - timestamps_m[k] + timestamps_m[i] - timestamps_m[k]$$

In the case of $n < k$ where indeterminacy could happen due to the missing of $k - 1^{st}$ local timestamp and beyond (this situation could happen when a recent fetch of the slave occurs later than the fetch of the master), the $delay[i]$ of the slave uses Equation (1) for $i \leq n$ and the $delay[i]$ of the slave for $n < i \leq k$ will be neglected as there is no appropriate local timestamps of the master that can be used for calculating the replication delay. The neglected calculations will be carried out later after the array of the master is updated.

In our analyser component instantiation and related SLA specification, the replication delay SLA for each replica ($delay_{sla}$) is defined as an integer value in the unit of millisecond which represents two main components:

$$delay_{sla} = delay_{rtt} + delay_{tolerance}$$

where the round-trip time component of the SLA replication delay ($delay_{rtt}$) is the average round-trip time from the master to the associated slave.

The tolerance component of the replication delay ($delay_{tolerance}$) is defined by a constant value which represents the tolerance limit of the period of the time for the replica to be inconsistent. This tolerance component can vary from one replica to another depending on many factor such as the application requirements, the geographic location of the replica, and the workload characteristics and the load balancing strategy of each application.

In our framework implementation, we follow an intuitive strategy that triggers the actuator component for adding a new replica when it detects a number of continuous up-to-date monitored replication delays of a replica which exceeds its application-defined threshold ($T$) of SLA violation of data freshness. In other words, for a running database replica, if the latest $T$ monitored replication delays are violating its SLA of data freshness, the analyser component will trigger the actuator component to activate the geographically closest replica (for the violating replica). It is worthy to note that the strategy of the analyser component in making the decisions (e.g. the timing, the placement, the physical creation) regarding the addition a new replica can play an important role in determining the overall performance of the framework. However, it is not the main focus of this paper to investigate different algorithms for making such decisions.

In our previous work, it has been noted that the effect of the configurations of geographic location of the database replica is not as significant as the effect of the overloading workloads in increasing the staleness window of the database replicas [2]. Therefore, the analyser component can decide to stop an active database replica when it detects a decreasing workload that can be served by less number of database replicas without violating the application-defined SLAs. This will reduce the monetary cost of the running application.

In the actuator component, “Starting a new slave replica” is used to bring a slave from a “paused” status to the “running” status. Once a slave is brought to alive, its local timestamps will be monitored, and replication delays will be tracked. Intuitively, a “paused” status is supposed to be stopped from both reading and writing, so that there is no user served and no master synchronised. However, after a few initial trials, we realised that an out-of-date slave, derived from the original copy of the master at the beginning of the experiment, takes time to synchronise after it is started. Moreover, if the slave is “too old”, the analyser component will trigger more “starting a new slave replica” actions as it detects SLA-violation in that slave during its synchronisation. One solution to this situation is to give tens of seconds for the slave to catch up with the master. However, the catch-up period cannot be fixed, the later the slave starts, the longer it takes to catch-up. Another solution is to create hot backups of the master frequently, so that all new slaves can be derived from a newest backup but we didn’t evaluate this solution in our experiment. In order to keep our evaluation experiments focused on the main benefits of our architecture framework, we simply relied on a set of hot backups which are originally not used for serving the application requests, but kept synchronised, and then can be turned for being active and used by the load balancer for serving the application requests when the actuator component is triggered for adding a new replica.

4.5 Experiment results

In order to measure the baselines of our comparison, we conducted two experiments without our framework in order to measure the replication delays of 3 (minimum number of running replicas) and 6 (maximum number of running replicas) slaves. For all experiments, we have set the value of the heartbeat interval ($intvl_{heartbeat}$) to 1 second and we set the value of the threshold for the maximum possible continuous SLA violations for any replica using the following formula: $T = \frac{intvl_{max}}{intvl_{heartbeat}}$.

Figure 3 illustrates the effect of our approach on the performance of the replication delay for the cloud-hosted database replicas. Figure (3a) and Figure (3b) show the replication delay of the two baseline cases for our comparison. They represent the experiments of running with a fixed number of replicas (3 and 6 respectively) from the starting times of the experiments to their end times. Figure (3a) shows that the replication delay tends to follow different patterns for the different replicas. The two trends of $us-west-1$ and $eu-west-1$ surge significantly at 260 and 280 users, respectively. On the same trend, the time of $us-east-1$ tends to be stable through out the entire running time of the experiment. The main reason behind that is the performance variation between the hosting EC2 instances for the database replicas. Due to the performance differences between the physical CPUs specifications, $us-east-1$ is able to handle the amount of operations that saturate $us-west-1$ and $eu-west-1$. Moreover, with an identical CPU for $us-west-1$ and $eu-west-1$, the

Both $us-west-1$ and $eu-west-1$ are powered by Intel(R) Xeon(R) CPU E5507 @ 2.27GHz, whereas $us-east-1$ is deployed with an better CPU, Intel(R) Xeon(R) CPU E5645 @ 2.40GHz.
former seems to surge at an earlier point than the latter. This is basically because of the difference in the geographical location of the two instances. As illustrated in Figure (3), the MySQL Proxy location is closer to us-west-1 than eu-west-1. Therefore, the forwarded database operations by the MySQL Proxy take less time to us-west-1 than to eu-west-1 which leads to more congestion on the us-west-1 side. Similarly, in Figure (3), the replication delay tends to surge in both us-west-1 and us-west-2 for the same reason of the difference in the geographic location of the underlying database replica.

Figure (3) shows the result of the replication delay for our experiments using 120 seconds for the monitor interval ($intvl_{mon}$) and 1000 milliseconds for the SLA of replication delay ($delay_{sla}$). The figure shows that the us-west-2, us-east-2, and eu-west-2 replicas are added in sequence at 255$^s$, 407$^s$ and 1843$^s$ seconds, where the drop lines are emphasized. The addition of the three replicas are caused by the SLA-violation of the us-west-1 replicas at different periods. In particular, there are four SLA-violation periods for us-west-1 where the period must exceed the monitor interval, and all calculated replication delays in the period must exceed the SLA of replication delay. These four periods are: 1) 67:415 (total of 349 seconds), 2) 670:841 (total of 172 seconds), 3) 1373:1579 (total of 207 seconds), 4) 1615:3000 (total of 1386 seconds). The adding of new replicas is only triggered on the 1$^{st}$ and the 4$^{th}$ periods based on the time point analysis. The 2$^{nd}$ and the 3$^{rd}$ periods do not trigger the addition of any new replica as the number of detected SLA violations does not exceed the defined threshold ($T$).

In general, the results of our experiments show that our approach can effectively reduce the replication delay of the underlying replicas by adding new replicas when necessary. It is also observed that with more replicas added, the replication delay for the overloaded replicas can dramatically drop. Moreover, it is more cost-effective in comparison to the over-provisioning approach for the number of database replicas that can ensure low replication delay because it adds new replicas only when necessary based on the application-defined SLA of data freshness for the different underlying database replicas. Both the proxy overhead and the framework overhead is low.

In addition, the architecture framework contains a new reusable monitoring infrastructure and highly customisable and extensible components. It will reduce future development cost for such scaling behaviours.

5. RELATED WORK

In our approach, we followed the architecture-based autonomic computing blueprint [7]. The blueprint was applied in other domains such as J2EE-based adaptive server applications [7] successfully. Our approach is the first attempt in the scaling of cloud-based relational database from a consumer-centric point of view.

In the context of consistency study, Kraska et al. [8] have presented an approach that allows developers to define the consistency guarantees at the data-level rather than the application level and has the ability to automatically switch consistency models at runtime. Keeton et al. [7] have proposed a similar approach in a system called LazyBase that allows users to trade off query performance and result freshness. While Consistency Rationing and LazyBase represent two approaches to adaptive consistency management for cloud-hosted databases, they are more targeting the perspective of cloud service providers. There is also a lack of a general reusable infrastructure and customisable components that could integrate consumer-centric performance monitoring data.

A common feature to the different cloud offerings of the cloud-provided storage systems and the cloud-provided database systems is the creation and management of multiple replicas of the stored data while a replication architecture is running behind-the-scenes to enable automatic failover management and ensure high availability of the service. In general, replicating for performance differs significantly from replicating for availability or fault tolerance. The distinction between the two situations is mainly reflected by the higher degree of replication, and as a consequence the need for supporting weak consistency when scalability is the motivating factor for replication [8]. In addition, the platform-supplied storage layer and the relational database as a service are mainly designed for multi-tenancy environment and they are more centric to the perspective of the cloud provider. Therefore, they do not provide support for any flexible mechanisms for scaling a single-tenant system (consumer perspective).

6. CONCLUSION

We presented an architecture framework for application-managed scaling of cloud-hosted virtualised relational databases. The architecture framework contains a new monitoring infrastructure, an XML-based policy language and customisable components for sensing, monitoring, analysing and actuation. It provides a consumer-centric support for such adaptive database scaling using application-specific policies. We evaluated the architecture framework with a modified Web 2.0 benchmark. The results show significant reduction of replication delays (thus improving performance) and monetary cost through autonomic scaling. The reusable infrastructure and architecture components also help the future development of a wide range of scaling behaviours and monitoring needs.

7. ACKNOWLEDGMENTS

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

This work is partially funded by AWS in Education Research Grants.

Figure 3: The performance of the adaptive management of the replication delay for the cloud-hosted database replicas.

(a) Fixed 3 running replicas
(b) Fixed 6 running replicas
(c) $\text{delay}_{\text{tolerance}} = 1000 \text{ ms}$ and $\text{intvl}_{\text{mon}} = 120\text{sec}$